EXHIBIT 7

Maker Docs

Transaction manager

The `transactionManager` service is used to track a transaction's status as it propagates through the blockchain.

Methods in Dai.js that start transactions are asynchronous, so they return promises. These promises can be passed as arguments to the transaction manager to set up callbacks when transactions change their status to `pending`, `mined`, `confirmed` or `error`.

```
const txMgr = maker.service('transactionManager');
// instance of transactionManager
const open = maker.service('cdp').openCdp();
// open is a promise--note the absence of `await`
```

Pass the promise to `transactionManager.listen` with callbacks, as shown below.

```
txMgr.listen(open, {
  pending: tx => {
    // do something when tx is pending
  },
  mined: tx => {
    // do something when tx is mined
  },
  confirmed: tx => {
    // do something when tx is confirmed
  },
  error: tx => {
    // do someting when tx fails
  }
});

await txMgr.confirm(open);
// 'confirmed' callback will fire after 5 blocks
```

Note that the `confirmed` event will not fire unless `transactionManager.confirm` is called. This async function waits a number of blocks (default 5) after the transaction has been mined to resolve. To change this globally, set the `confirmedBlockCount` attribute in Maker options. To change it for just one call, pass the number of blocks to wait as the second argument:

```
await txMgr.confirm(open, 3);
```

Transaction Metadata

There are functions such as `lockEth()` which are composed of several internal transactions. These can be more accurately tracked by accessing `tx.metadata` in the callback which contains both the `contract` and the `method` the internal transactions were created from.

Transaction Object Methods

A `TransactionObject` also has a few methods to provide further details on the transaction:

- `hash` : transaction hash
- `fees()` : amount of ether spent on gas
- `timeStamp()` : timestamp of when transaction was mined
- `timeStampSubmitted()` : timestamp of when transaction was submitted to the network

```
const lock = cdp.lockEth(1);
txMgr.listen(lock, {
  pending: tx => {
    const {contract, method} = tx.metadata;
    if(contract === 'WETH' && method === 'deposit') {
      console.log(tx.hash); // print hash for WETH.deposit
    }
  }
})
```

Maker Docs

Vault manager

The vault manager works with vaults that are owned by the CdpManager contract, which is also used by Oasis Borrow. This intermediary contract allows the use of incrementing integer IDs for vaults, familiar to users of Single-Collateral Sai, as well as other conveniences.

In the code, this is called CdpManager.

```
const mgr = maker.service('mcd:cdpManager');
```

Instance methods

The methods below are all asynchronous.

getCdpIds()

Return an array describing the vaults owned by the specified address. Note that if the vaults were created in Oasis Borrow, the address of the proxy contract should be used.

```
const proxyAddress = await maker.service('proxy').currentProxy();
const data = await mgr.getCdpIds(proxyAddress);
const { id, ilk } = data[0];
// e.g. id = 5, ilk = 'ETH-A'
```

getCdp()

Get an existing vault by its numerical ID. Returns a Vault instance.

```
const vault = await mgr.getCdp(111);
```

open()

Open a new vault with the specified collateral type. Will create a proxy if one does not already exist. Returns a Vault instance. Works with the transaction manager.

```
const txMgr = maker.service('transactionManager');
const open = await mgr.open('ETH-A');
txMgr.listen(open, {
4
```

```
5 }),pending: tx => console.log('tx pending: ' + tx.hash)
6 const vault = await open;
```

openLockAndDraw()

Open a new vault, then lock and/or draw in a single transaction. Will create a proxy if one does not already exist. Returns a Vault instance.

```
1 const vault = await mgr.openLockAndDraw(
2   'BAT-A',
3   BAT(1000),
4   DAI(100)
5 );
```

Vault instances

In the code, these are called ManagedCdp.

Properties

A note on caching: When a vault instance is created, its data is pre-fetched from the blockchain, allowing the properties below to be read synchronously. This data is cached in the instance. To refresh this data, do the following:

```
1 vault.reset();
2 await vault.prefetch();
```

collateralAmount

The amount of collateral tokens locked, as a currency unit.

collateralValue

The USD value of collateral locked, given the current price according to the price feed, as a currency unit.

debtValue

The amount of Dai drawn, as a currency unit.

liquidationPrice

The USD price of collateral at which the Vault becomes unsafe.

isSafe

Whether the Vault is currently safe or not.

Instance methods

All of the methods below are asynchronous and work with the transaction manager. Amount arguments should be currency units, e.g.:

```
import { ETH, DAI } from '@makerdao/dai-plugin-mcd';

await vault.lockAndDraw(ETH(2), DAI(20));
```

lockCollateral(amount)

Deposit the specified amount of collateral.

drawDai(amount)

Generate the specified amount of Dai.

lockAndDraw(lockAmount, drawAmount)

Deposit some collateral and generate some Dai in a single transaction.

wipeDai(amount)

Pay back the specified amount of Dai.

wipeAll()

Pay back all debt. This method ensures that dust amounts do not remain.

freeCollateral(amount)

Withdraw the specified amount of collateral.

wipeAndFree(wipeAmount, freeAmount)

Pay back some debt and withdraw some collateral in a single transaction.

wipeAllAndFree(freeAmount)

Pay back all debt, ensuring dust amounts do not remain, and withdraw a specified amount of collateral in a single transaction.

give(address)

Transfer ownership of this vault to `address`. Note that if the new owner plans to use this vault with Oasis Borrow, it should be transferred to their proxy with giveToProxy instead.

giveToProxy(address)

Look up the proxy contract owned by `address` and transfer ownership of this vault to that proxy.